1. **Inheritance**
   1. Characteristics :
      1. Capability of a class to derive properties and characteristics from another class
      2. One of the most important features of OOP.
      3. **Derived** class(Sub/Child Class) inherits all the properties of the **base** class(Super/Parent)
      4. No change in the properties of base class
      5. May add new features to its own.
      6. New features in the derived class will not affect the base class
      7. Derived class is a specialised base class
   2. Advantages :
      1. Inheritance helps the code to be **reused** in many situations.
      2. Programmer can :
         1. create as many derived classes from the base class as needed
         2. add specific features to each derived class as needed.
      3. Saves the programmer Time and Effort.
      4. Method Overriding (Hence, Runtime Polymorphism.)
      5. Use of Virtual Keyword
   3. The visibility modes control the access-specifier to be for inheritable members of base-class, in the derived class.
      1. Public Mode
         1. Attributes and methods are accessible from any part of the program, including classes that derive from the base class.
         2. Inheritance allows derived classes to access and use public members of the base class.
      2. Private Mode
         1. Attributes and methods are only accessible within the class in which they are defined.
         2. Derived classes cannot access private members of the base class(are entirely **encapsulated** [refers to the practice of bundling data (attributes) and the methods (functions) that operate on that data into a single unit called a class.] within the base class.)
      3. Protected Mode
         1. Attributes and methods are accessible within the class and its subclasses (derived classes).
         2. Derived classes can access protected members of the base class, but code outside the class hierarchy cannot directly access them.
   4. Syntax :- **class** subclass : **AccessSpecifier** superclass
2. **Single Inheritance :**
   1. Superclass -> subclass
   2. Syntax:

class Classname // base class

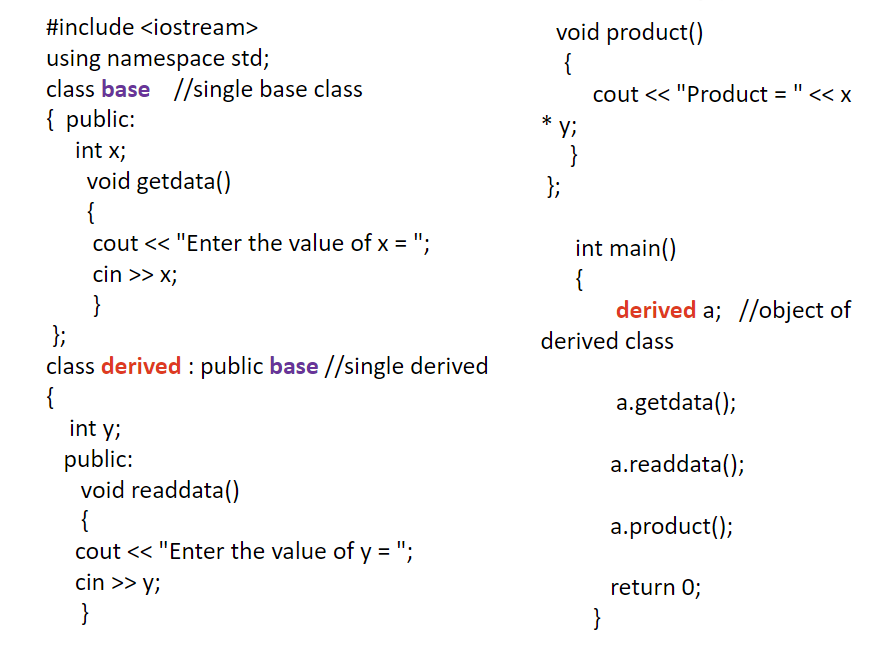
{

};

class classname: access\_specifier baseclassname

{

};



* 1. Applications :
     1. University Grading System
     2. Employee and Salary

1. **Multiple Inheritance :** 
   1. Subclass1 ->

Superclass

Subclass2 ->

* 1. Syntax:

class A // base class

{

};

class B

{

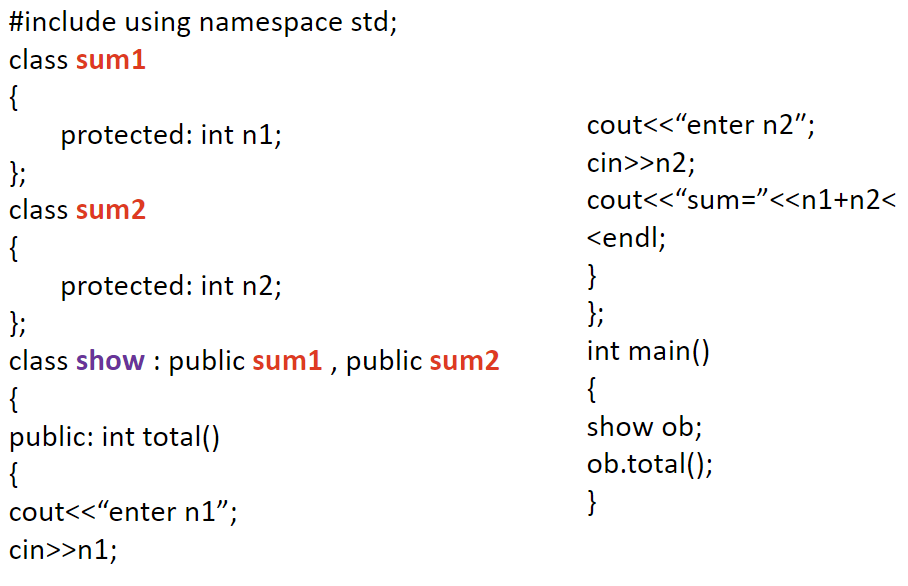
}

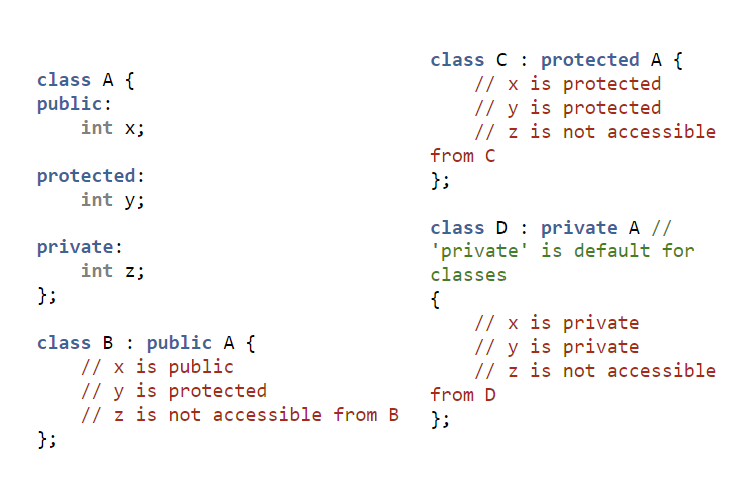
class c : access\_specifier A, access\_specifier B // derived class

{

} ;

* 1. Applications - Distributed Database





1. **Multilevel inheritance :**
   1. Parent

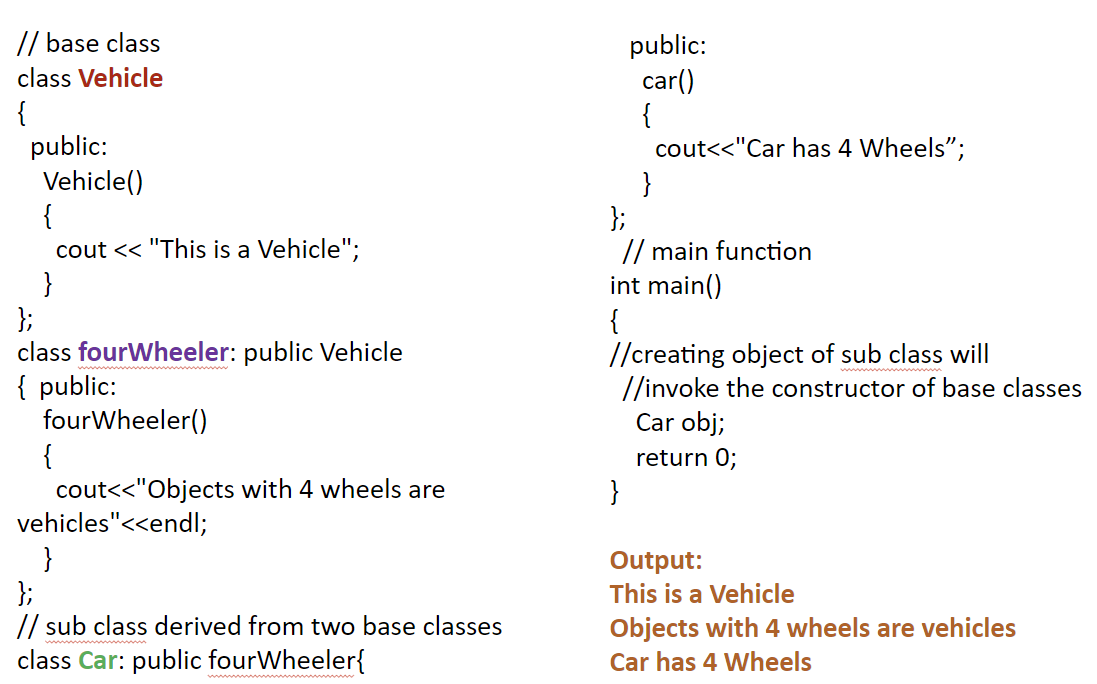
|

V

Child 1

|

V

Child 2

1. **Hierarchical**
   1. Parent

| | |

V V V

A B C

#include <iostream>

using namespace std;

class A **//single base class**

{

public:

int x, y;

void getdata()

{

cout << "\nEnter value of x and y:\n";

cin >> x >> y;

}

};

class B : public A **//B is derived from class base**

{

public:

void product()

{

cout << "\nProduct= " << x \* y;

}

};

class C : public A **//C is also derived from class base**

{

public:

void sum()

{

cout << "\nSum= " << x + y;

}

};

int main()

{

B obj1;  **//object of derived class B**

C obj2; **//object of derived class C**

obj1.getdata();

obj1.product();

obj2.getdata();

obj2.sum();

return 0;

}

1. **Hybrid** 
   1. Parent

| |

V V

A B

|

V

Child 2

* 1. Eg

class A

{

public:

int x;

};

class B : public A

{

public:

B()

{

x = 10;

}

};

class C

{ public:

int y;

C()

{

y = 4;

}

};

class D : public B, public C

{ public:

void sum()

{

cout << "Sum= " << x + y;

}

};

1. **Order of Constructor Call**
   1. create derived class object
      1. base class default constructor is executed
      2. derived class's constructor finishes execution.
   2. base class's **default** constructor is always called when derived class's
      1. Default
      2. Parameterised

constructor is called

* 1. base class's **parameterised** constructor
     1. When derived class's parameterised constructor is declared
     2. we must mention base class's **parameterised** constructor

explicitly

**Why all the constructors are called?**

* 1. Difference between Derived class
     1. constructor has access only to its own class members
     2. object also have inherited property of Base class
  2. only base class constructor can properly initialize base class members.

1. **Constructors and Destructors** 
   1. are never inherited
   2. Hence never overridden.
   3. Also, assignment operator = is never inherited. It can be overloaded but can't be inherited by sub class.
   4. **Static** Function :
      1. inherited into the derived class.
      2. Can be redefined in a derived class
      3. [ all the other overloaded functions in base class are hidden.]
      4. can **never** be **virtual**.
2. **Derived class** 
   1. can inherit all base class methods
   2. cannot inherit
      1. Constructors
      2. Destructors
      3. Copy constructors
      4. Overloaded operators
      5. Friend functions

**//Calling base and derived class method using base reference**

#include <iostream>

using namespace std;

class Foo

{public:

int x;

virtual void printStuff()

{

cout<<"BaseFoo printStuff called"<<endl;

}

};

class Bar : public Foo

{

public:

int y;

void printStuff()

{cout<<"derived Bar printStuff called"<<endl;

}

};

int main()

{Foo \*foo=new Foo;

foo->printStuff();/

**//this call the base function**

foo=new Bar;

foo->printStuff();

}

Output:

Base Foo printStuff called

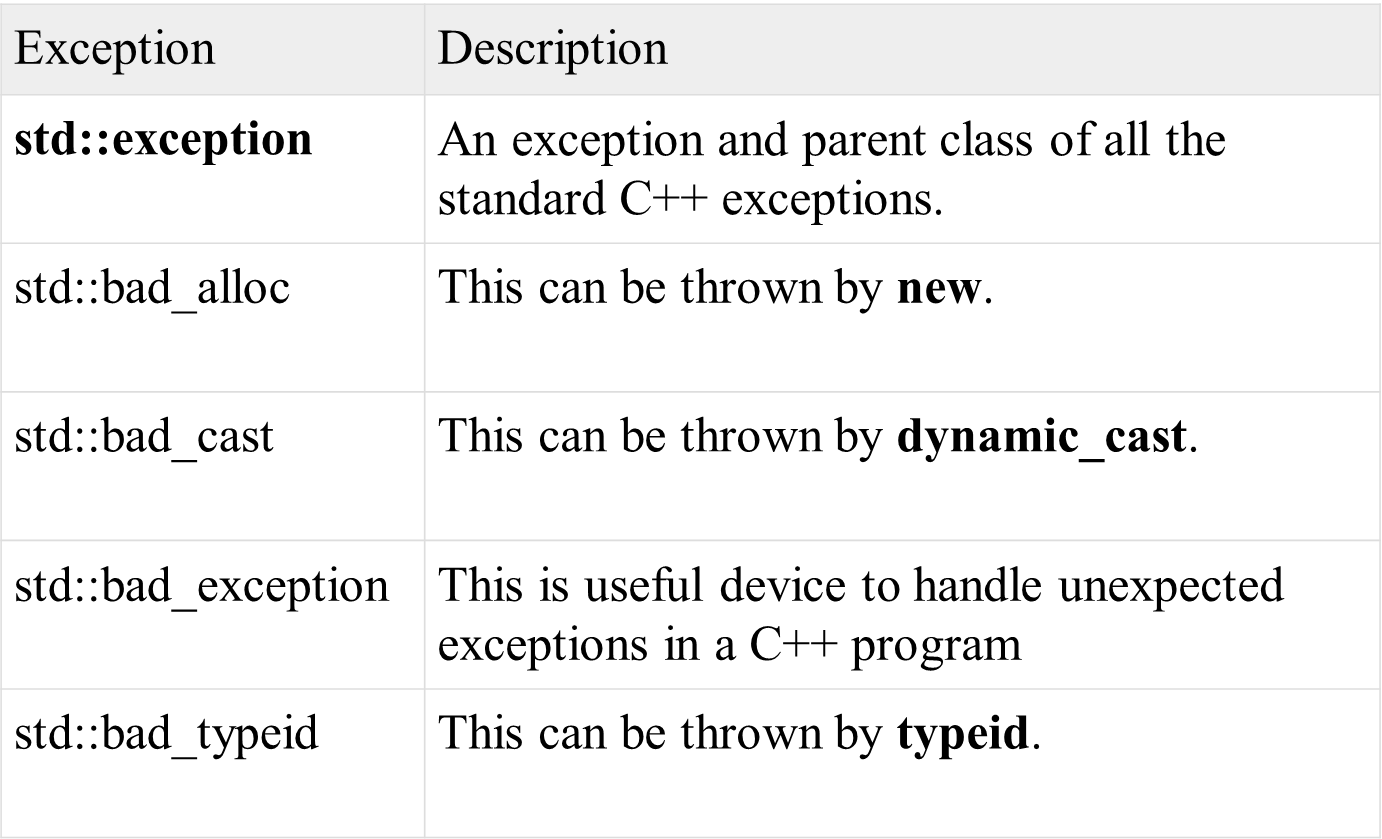
derived Bar printStuff called

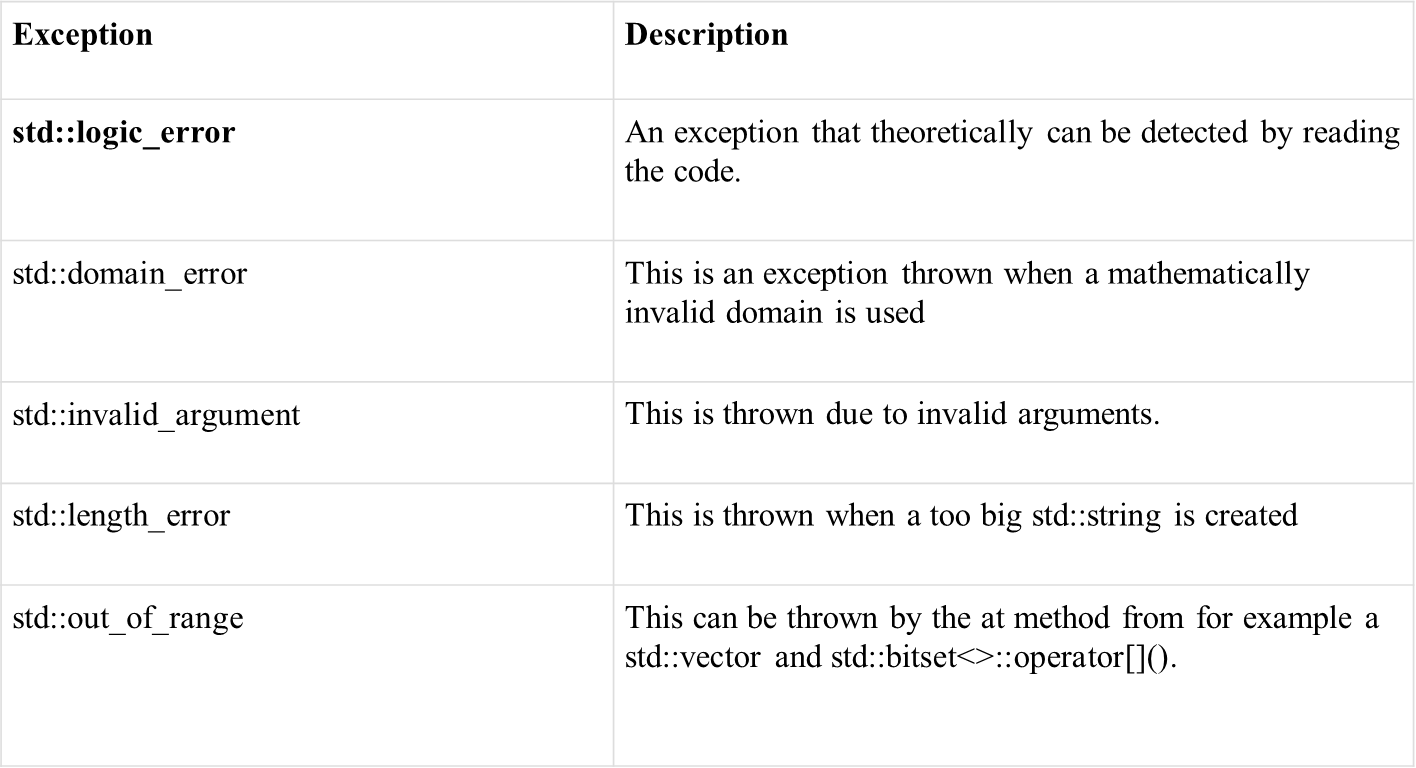
1. **Advanced Functions**
   1. **Inline** 
      1. **Implicit**
      2. **Explicit**
   2. **Friend**
   3. **Virtual Function**
      1. Declared in base class
      2. **overridden** in the derived class,
      3. tells the compiler to perform **Late Binding** on this function.
      4. make a member function of the base class virtual
      5. allows the most specific version of a member function in an inheritance hierarchy to be selected for execution.
      6. make polymorphism possible.
   4. **Pure virtual function**
      1. hv no definition
      2. start with virtual keyword
      3. ends with = 0
      4. a small definition in the Abstract class
      5. cannot create object of Abstract class.
      6. **must be defined outside the class definition**
      7. **Inline** pure virtual definition is **Illegal**.
2. **Overriding**
   1. Advantages of **Function Overriding**
      1. improve the
         1. **Readability**
         2. **Consistency**
         3. **Reusability**
         4. **Clean code**

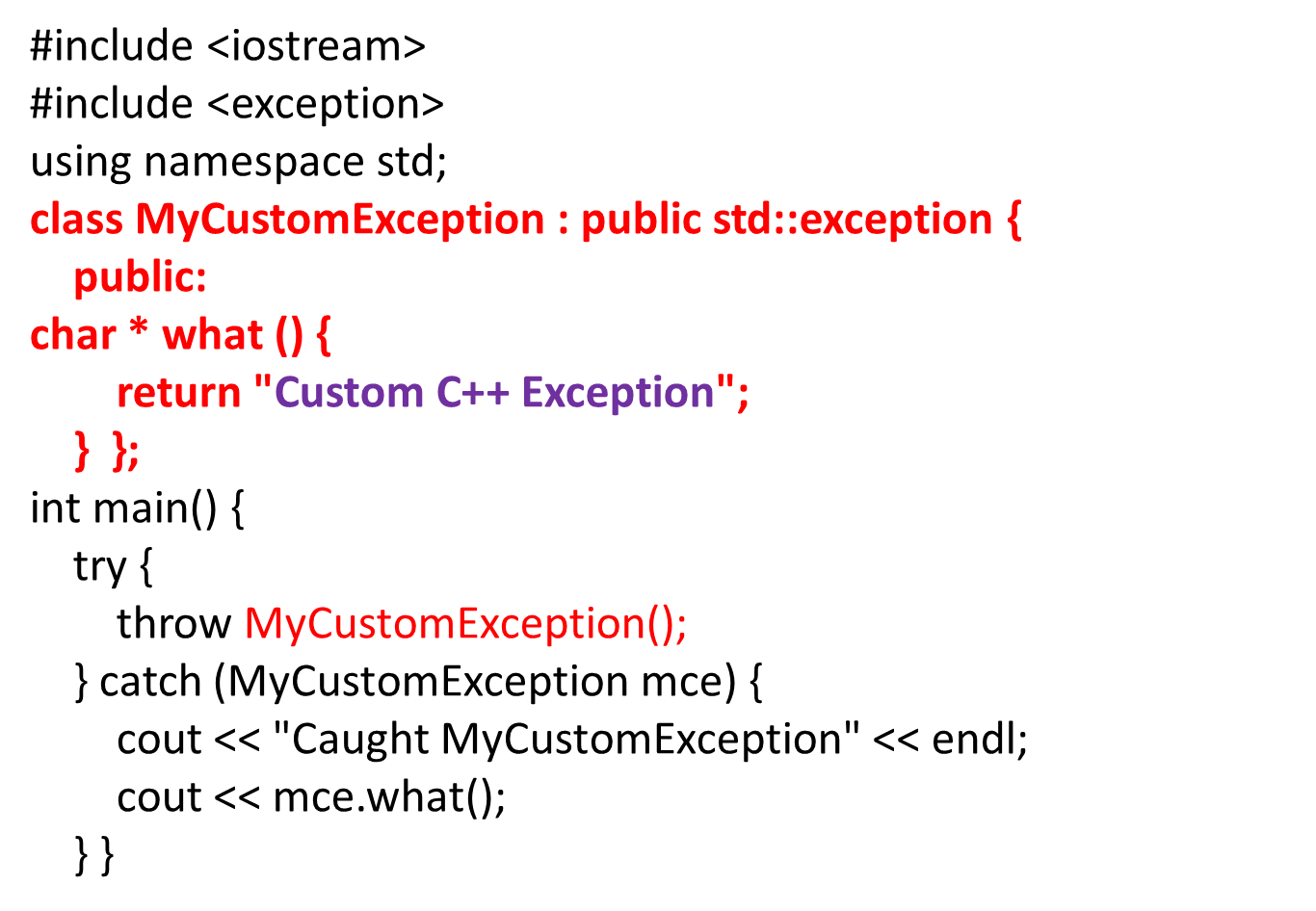
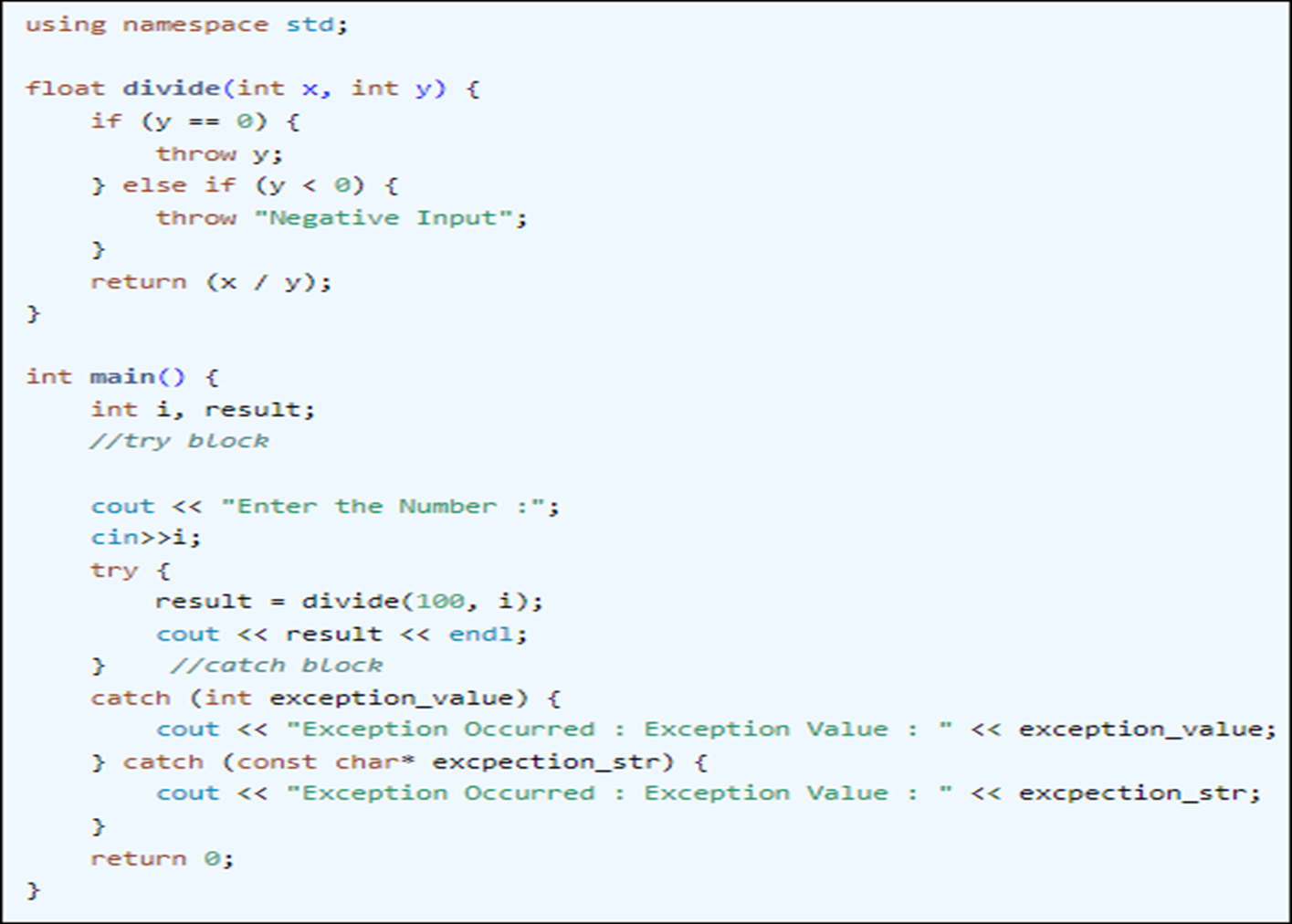
of code

* + 1. If both the child class and base class have the same function, it **will not affect the independence of the child class** function.
    2. saves the memory
    3. A function with the same name can be used to perform different operations

1. **Abstract class**
   1. is a class which contains **atleast one Pure** Virtual function
   2. used to provide an Interface for its sub classes
   3. Classes inheriting an Abstract Class must provide definition to the pure virtual function
   4. otherwise they will also become abstract class.
   5. Characteristics:
      1. cannot be instantiated
      2. pointers and references can be created.
      3. can have normal functions and variables
      4. mainly used for Upcasting (Upcasting is using the Super class's reference or pointer to refer to a Sub class's object.), so that its derived classes can use it interface.
2. **Interface**
   1. describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class
   2. implemented using abstract classes
   3. **Signature :** return\_type method\_name(values/parameters)
   4. set of all signatures defined by an object’s methods
   5. abstract class that cannot be instantiated
   6. Explicitly implementing interface in a class enables us to define a set of methods that are mandatory for that class
   7. When to use Interface?
      1. an immutable contract is really intended
      2. your applications require many possibly unrelated object types to provide certain functionality
      3. do not need to inherit implementation from a base class.
      4. Like multiple inheritance
3. **UML State Chart Diagram**
   1. To model object states of a system.
   2. To model reactive system. Reactive system consists of reactive objects.
   3. To identify events responsible for state changes.
   4. Forward and reverse engineering.
   5. Identify
      1. important objects to be analysed.
      2. states.
      3. events.
4. **UML Activity Diagram**
   1. logic of an algorithm.
   2. It describes the functions performed in use cases.
   3. Illustrate a business process or **workflow** between users and the system.
   4. Simplifies and improves any process by **descriptive complex use cases**.
   5. Model software architecture elements, such as method, function, and operation.
   6. sequential or concurrent
   7. sequence of actions or flow of control in a system
5. **Templates**
   1. Allows functions and classes to operate with generic types.
   2. Allows a function or class to work on many **different data types without being rewritten** for each one.
   3. Great utility when combined with **multiple inheritance and operator overloading**
   4. The C++ Standard Library is based upon conventions introduced by the **Standard Template Library (STL)**
   5. **Types:**
      1. **Function Template:** 
         1. behaves like a function except that the template can have arguments of many different types
         2. A single function template can work on different types at once but, different functions are needed to perform identical task on different data types
         3. Why Function Templates?
            1. **instantiated** at **compile**-**time** with the source code.
            2. **less** code than overloaded C++ functions.
            3. **type safe.**
            4. allow user-defined specialization.
            5. allow non-type parameters.
      2. **Class Template:** 
         1. provides a specification for generating classes based on parameters.
         2. Class templates are generally used to implement containers.
6. **Exceptions:**
   1. **types:**
      1. **Hardware/operating system level.** 
         1. Arithmetic exceptions; divide by 0.
         2. Memory access violations; stack over/underflow.
      2. **Language level.** 
         1. Type conversion; illegal values, improper casts.
         2. Bounds violations; illegal array indices.
         3. Bad references; null pointers.
      3. **Program level**.
         1. User defined exceptions.
   2. **Mechanism**
      1. Find the problem (Hit the exception)
      2. Inform that an error has occurred (Throw the exception)
      3. Receive the error information (Catch the exception)
      4. Take corrective actions (Handle the exception)

****

****

****